# 4 EXPLOITING SOFTWARE DIVERSIFICATION FOR WEBASSEMBLY

In this chapter we instantiate the usage of Software Diversification for offensive and defensive purposes. We present two selected use cases that exploit Software Diversification through our technical contributions presented in Chapter 3. The selected cases are representative of applications of Software Diversification for WebAssembly in browsers and standalone engines.

## 4.1 Offensive Diversification: Malware evasion

The primary malicious use of WebAssembly in browsers is cryptojacking [**?** ]. This is due to the essence of cryptojacking, the faster the mining, the better. Although the research of Lehmann and colleagues [**?** ] suggests a decline in browser-based cryptominers, mainly due to the shutdown of Coinhive, a 2022 report by Kaspersky indicates that the use of cryptominers is on the rise [**?** ]. This underscores the ongoing need for effective automatic detection of cryptojacking malware.

Both antivirus software and browsers have implemented measures to detect cryptojacking. For instance, Firefox employs deny lists to detect cryptomining activities [**?** ]. The academic community has also contributed to the body of work on detecting or preventing WebAssembly-based cryptojacking, as outlined in Subsection 2.1.6. However, it's worth noting that malicious actors can employ evasion techniques to circumvent these detection mechanisms. Bhansali et al. are among the first who have investigated how WebAssembly cryptojacking could potentially evade detection [**?** ], highlighting the critical importance of this use case. For an in-depth discussion on this topic, we direct the reader to our

[0] Comp. time 2023/10/06 06:14:24

contribution [**?** ]. The use of case illustrated in the subsequent sections uses Offensive Software Diversification for the sake of evading malware detection in WebAssembly.

**TODO** Replace by a time diagram.

### 4.1.1 Threat model: cryptojacking

Let us illustrate the threat model in which a malicious Wasm binary could be involved. Figure 4.1 illustrates a browser attack scenario: a practical WebAssembly cryptojacking attack consists of three components: a WebAssembly binary, a JavaScript wrapper, and a backend cryptominer pool. The WebAssembly binary is responsible for executing the hash calculations, which consume significant computational resources. The JavaScript wrapper facilitates the communication between the WebAssembly binary and the cryptominer pool.

For the previous triad to work, the following steps are executed. First, the victim visits a web page infected with the cryptojacking code. The web page establishes a channel to the cryptominer pool, which then assigns a hashing job to the infected browser. The WebAssembly cryptominer calculates thousands of hashes inside the browser. Once the malware server receives acceptable hashes, it is rewarded with cryptocurrencies for the mining. Then, the server assigns a new job, and the mining process starts over.

### 4.1.2 Approach

Several techniques, as outlined in Subsection 2.1.6, can be directly implemented in browsers to thwart cryptojacking by identifying the malicious WebAssembly components. The primary aim of this use of case is to investigate the effectiveness of code diversification as a means to circumvent cryptojacking defenses. Specifically, we assess whether the following evasion workflow can successfully bypass existing security measures:

1. The user lands on a webpage infected with cryptojacking malware, which leverages network resources for execution—corresponding to ① and ② in Figure 4.1. Notice that, various methods can be used to inject cryptojacking malware, including malicious browser extensions, malvertising, compromised websites, or deceptive links [**?** ].

2. A malware detection mechanism identifies and blocks malicious WebAssembly binaries at ③. For example, a network proxy could intercept and forward these resources to an external detection service via its API.

3. Anticipating that a specific malware detection system is consistently used for defense, the attacker swiftly generates a variant of the WebAssembly cryptojacking malware designed to evade detection at ④.

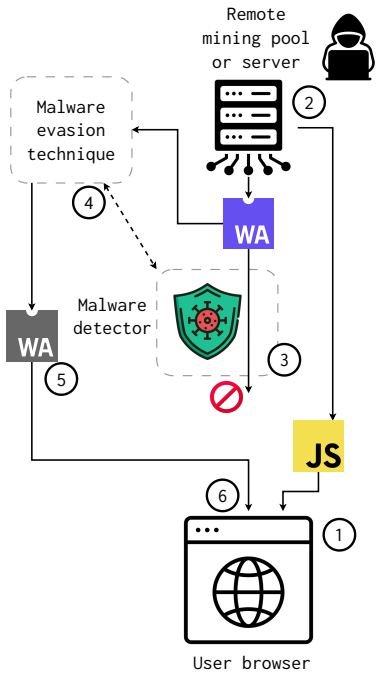


***Figure 4.1***

4. The attacker delivers the modified binary instead of the original one (5), which initiates the cryptojacking process and compromises the browser (6). The detection method is completely oblivious to the malicious nature of the binary, and the attack is successful.

To empirically validate this evasion scenario, we conducted experiments on 33 cryptojacking malware samples, curated from the 8643 binaries in the wasmbench dataset [**?** ]. These 33 binaries were flagged as potentially hazardous by at least one antivirus vendor on VirusTotal. We employed WASM-MUTATE to simulate (4) in Figure 4.1. Our evaluation focuses on three key metrics: the success rate of evading VirusTotal's detection mechanisms across the 33 binaries, the speed at which WASM-MUTATE generates a detection-evasive variant, and the performance impact on the variants that successfully evade the detection.

### 4.1.3 Results

**TODO** Uncontrolled and controlled diversification

**TODO** How many iterations are needed to evade the detection? Compare with the table in chapter 3, 10000 variants in one hour is equivalent to say that we can evade the detection in 10 minutes.

**TODO** Talk about performance. Highlight the presence of bloating...the faster you compile, the faster you start ahasing. Link this with the paper that Marin shared lately, highlighting the lack of optimization.

Remarkably, we find 30 cryptominers for which our technique successfully generates variants that evade VirusTotal. Our set of malware includes 6 cryptojacking programs that are fully reproducible in a controlled environment. With them, we assess that our evasion method does not affect malware correctness and generates fully functional malware variants with minimal overhead.

Our work provides evidence that the malware detection community has opportunities to strengthen the automatic detection of cryptojacking WebAssembly malware. The results of this work are actionable, as we provide quantitative evidence on specific malware transformations on which detection methods can focus.

Contribution paper

The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WebAssembly Diversification for Malware Evasion" *at Computers & Security, 2023* `https://www.sciencedirect.com/science/article/pii/S0167404823002067`.

## 4.2 Defensive Diversification: Speculative Side-channel protection

As previously discussed in Subsection 2.1.5, WebAssembly is rapidly gaining traction in backend environments. Companies like Cloudflare and Fastly are encouraging the use of WebAssembly in their edge computing platforms, allowing developers to deploy faster applications in a modular and securely sandboxed fashion. Typically, these client WebAssembly applications are designed as isolated services with a single, focused responsibility. This model is usually called Function-as-a-Service (FaaS) [**? ?** ].

The core concept behind Wasm in FaaS platforms is the ability to host thousands of client WebAssembly binaries within a single host process, which is then distributed across multiple servers and data centers. To achieve this, the platforms compile the Wasm programs to native code, which is then executed in a sandboxed environment. Then, the host processes are capable of instantiating a new Wasm sandbox for each client function, executing it in response to individual user requests in a matter of nanoseconds. Utilizing WebAssembly enables these platforms to inherently isolate the execution of client functions from one another

as well as from the host process. However, this isolation is not foolproof against Spectre attacks [**?** **?** ].

In the subsequent sections, we illustrate how diversification can be used to protect WebAssembly binaries against Spectre attacks. We show a case of Defensive Software Diversification for the sake of protecting WebAssembly binaries.

### 4.2.1 Threat model: speculative side-channel attacks

Let us illustrate the threat model in which a WebAssembly program could be susceptible in FaaS platforms. Developers can submit any WebAssembly binary to the FaaS platform. This includes potential malicious actors that could upload a Wasm binary that, when compiled to native code, uses Spectre attacks to leak sensitive information from the host process. Spectre attacks exploit hardware predictors to induce misspredictions and speculatively execute instructions—gadgets that would not run sequentially. The attacker could then use this information to infer the contents of the memory of other client functions, or even the host process itself.

Narayan and colleagues [**?** ] dissected the possible Spectre attacks for Wasm binaries into three categories based on the specific hardware predictor that is exploited and the specific FaaS scenario: Sandbox breakout attacks, Sandbox poisoning attacks and Host poisoning attacks. The first one, exploits the branch target buffer by predicting the target of an indirect jump, thereby rerouting speculative control flow to an arbitrary target. The second one takes advantage of the pattern history table to anticipate the direction of a conditional branch during the ongoing evaluation of a condition. The third and last one exploits the return stack buffer that stores the locations of recently executed call instructions to predict the target of `ret` instructions. Each attack methodology relies on the extraction of memory bytes from another hosted Wasm binary that executes in the same host process.

**TODO** Add diagram **TODO** Explain here the threat model with the diagram

### 4.2.2 Approach

- Use of wasm-mutate

### 4.2.3 Results

- Diminshing of BER - Rockiki paper on portable side channel in browsers.

**TODO** TBD discuss deoptimization

### 4.2.4 Partial input/output validation

When WASM-MUTATE generates a variant, it can be executed to check the input/output equivalence. If the variant has a _start function, both binaries, the original and the variant can be initialized. If the state of the memory, the globals and the stack is the same after executing the _start function, they are partially equivalent.

The _start function is easier to execute given its signature. It does not receive parameters. Therefore, it can be executed directly. Yet, since a WebAssembly program might contain more than one function that could be indistinctly called with and arbitrary number of parameters, we are not able to validate the whole program. Thus, we call the checking of the initialization of a Wasm variant, a partial validation.

### 4.2.5 Some other works to be cited along with the paper. Mostly in the Intro

*Spectre and side-channel defenses*

- paper 2021: Read this, since it is super related, `https://www.isecure-journal.com/article_136367_a3948a522c7c59c65b65fa87571fde7b.pdf` [**?** ]

- A dataset of Wasm programs: [**?** ]

- Papers 2020

- Papers 2019 - [**?** ]

Selwasm: A code protection mechanism for webassembly

Babble

- https://arxiv.org/pdf/2212.04596.pdf

Principled Composition of Function Variants for Dynamic Software Diversity and Program Protection

- https://dl.acm.org/doi/10.1145/3551349.3559553

How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes

- https://ieeexplore.ieee.org/document/9975423

Code obfuscation against symbolic execution attacks

Code artificiality: A metric for the code stealth based on an n-gram model

Semantics-aware obfuscation scheme prediction for binary

Wobfuscator: Obfuscating javascript malware via opportunistic translation to webassembly

Synthesizing Instruction Selection Rewrite Rules from RTL using SMT "We also synthesize integer rewrite rules from WebAssembly to RISC-V "

Wafl: Binary-only webassembly fuzzing with fast snapshots

Contribution paper

The case discussed in this section is fully detailed in Cabrera-Arteaga et al. "WASM-MUTATE: Fast and Effective Binary Diversification for WebAssembly" *Under review* `https://arxiv.org/pdf/2309.07638.pdf`.

## 4.3 Conclusions